CS2100 Computer Organisation

C Programming

- `int sumArray(int [], int);`
  `int sumArray(int arr[], int size);`

- `fgets(str, size, stdin) // reads size-1 chars,`
  `// or until '\n' (then output will have '\n')`
  `scanf("%s", str); // reads until white space`
  `puts(str); // terminates with newline`
  `printf("%s\n", str);`

- **Operator precedence**:

Operator	Assoc
`expr++ expr-- () [] . ->`	L to R
`++expr --expr ! ~ (type) * & sizeof`	R to L
`* / %`	L to R
`+ -`	L to R
`<< >>`	L to R
`< <= > >=`	L to R
`== !=`	L to R
`&`	L to R
∧	L to R
|	L to R
`&&`	L to R
||	L to R
`?:`	R to L
`= += -= *- /= %= <<= >>= &= ∧= |=`	R to L
`,`	L to R

- **ASCII values**:

Char	Dec	Hex	Bin
'0'	48	0x30	0b00110000
'A'	65	0x41	0b01000001
'a'	97	0x61	0b01100001

- **Nice numbers**:
  $2^{15}-1 = 32\,767$
  $2^{16}-1 = 65\,535$
  $2^{31}-1 = 2\,147\,483\,647$
  $2^{32}-1 = 4\,294\,967\,295$

Number Formats

Integer Formats

- **Sign extension for fixed-point numbers**:
  1's complement: extend sign bit to both left and right
  2's complement: extend sign bit to left and zeroes to right

- **Addition**:
  Perform binary addition. For 1's complement, add the carry-out of MSB to LSB. For 2's complement, ignore carry-out of MSB. If A and B have the same sign but result has opposite sign, overflow occurred. Additionally for 2's complement, if carry-in to MSB ≠ carry-out of MSB, overflow has occurred.

IEEE 754 Floating Point Format

	MSB ⟵ ⟶ LSB		
	Sign	Exponent	Mantissa
Single-precision	**1** bit	**8** bits *(excess-127)*	**23** bits
Double-precision	**1** bit	**11** bits *(excess-1023)*	**52** bits

Instruction Set Architecture

- **Big-endian**: Most sig. byte in lowest address. (MIPS)
  Little-endian: Least sig. byte in lowest address. (x86)

- **Complex Instruction Set Computer (CISC)**
  - Single instruction performs complex operation
  - Smaller program size as memory was premium
  - Complex implementation, no room for hardware optimization

  Reduced Instruction Set Computer (RISC)
  - Keep the instruction set small and simple, makes it easier to build/optimize hardware
  - Burden on software to combine simpler operations to implement high-level language statements

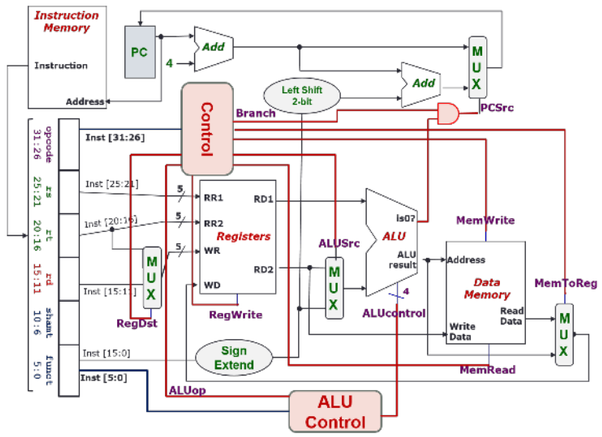
- **Stack architecture**
  - Operands are implicitly on top of the stack.

  Accumulator architecture
  - One operand is implicitly in the accumulator register.

  General-purpose register architecture
  - Only explicit operands.
  - Register-memory architecture (one operand in memory).
  - Register-register (or load-store) architecture.

  Memory-memory architecture
  - All operands in memory. Example: DEC VAX.

The Processor



Datapath & Control

- **Datapath**: Collection of components that process data; performs the arithmetic, logical and memory operations
- **Control**: Tells the datapath, memory and I/O devices what to do according to program instructions

Instruction Execution Cycle

1. Instruction Fetch: Get instruction from memory using address from PC register
2. Instruction Decode: Find out the operation required
3. Operand Fetch: Get operands needed for operation
4. Execute: Perform the required operation
5. Result Write: Store the result of the operation

MIPS combines Decode and Operand Fetch;
MIPS splits Execute into ALU and Memory Access

Clock

- PC is read during the first half of the clock period and it is updated with PC+4 at the

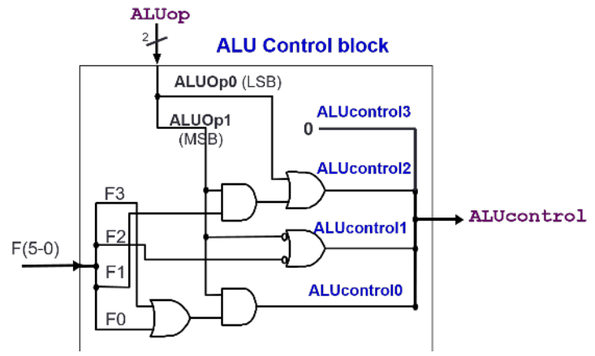
Control Signals

`RegDst`	Decode/Operand Fetch	Select the destination register number
`RegWrite`	Decode/Operand Fetch; RegWrite	Enable writing of register
`ALUSrc`	ALU	Select the 2nd operand for ALU
`ALUControl`	ALU	Select the operation to be performed
`MemRead` / `MemWrite`	Memory	Enable reading/writing of data memory
`MemToReg`	RegWrite	Select the result to be written back to register file
`PCSrc`	Memory/RegWrite	Select the next PC value

ALUOp Signal (2-bits)

`lw` & `sw`	`00`
`beq`	`01`
R-type	`10`

ALU Control Unit



ALUControl Signal (4-bits, MSB → LSB)

`Ainvert` (1 bit)	Whether to invert 1st operand
`Binvert` (1 bit)	Whether to invert 2nd operand
`Operation` (2 bits)	`00` AND | `01` OR | `10` add | `11` slt

Boolean Algebra

Laws & Theorems

Identity	$A+0=0+A=A$	$A\cdot 1=1\cdot A=A$
Inverse/complement	$A+A'=1$	$A\cdot A'=0$
Commutative	$A+B=B+A$	$A\cdot B=B\cdot A$
Associative	$A+(B+C)=(A+B)+C$	$A\cdot(B\cdot C)=(A\cdot B)\cdot C$
Distributive	$A\cdot(B+C)=(A\cdot B)+(A\cdot C)$	$A+(B\cdot C)=(A+B)\cdot(A+C)$

Idempotency	$X+X=X$	$X\cdot X=X$
One element / Zero element	$X+1=1$	$X\cdot 0=0$
Involution	$(X')'=X$	
Absorption	$X+X\cdot Y=X$	$X\cdot(X+Y)=X$
Absorption (var.)	$X+X'\cdot Y=X+Y$	$X\cdot(X'+Y)=X\cdot Y$
De Morgan's	$(X+Y)'=X'\cdot Y'$	$(X\cdot Y)'=X'+Y'$
Consensus	$X\cdot Y+X'\cdot Z+Y\cdot Z=X\cdot Y+X'\cdot Z$	$(X+Y)\cdot(X'+Z)$

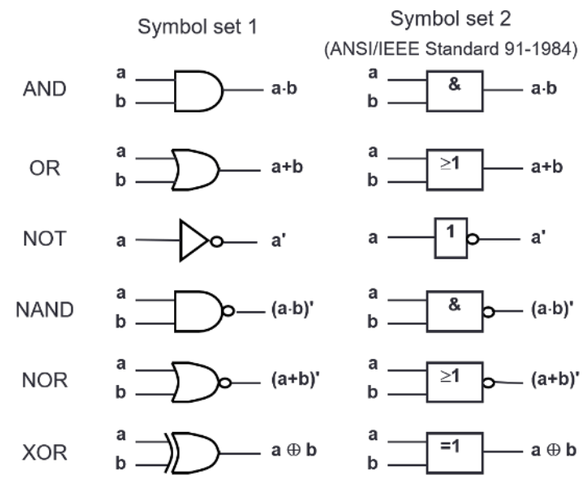
Minterms & Maxterms

- **Minterm**: $m0 = X'\cdot Y'\cdot Z'$
- **Maxterm**: $M0 = X+Y+Z$
- $m0' = M0$
- **Sum of minterms**: $\sum m(0,2,3) = m0+m2+m3$
- **Product of maxterms**: $\prod M(0,2,3) = M0\cdot M2\cdot M3$
- $\sum m(1,4,5,6,7) = \prod M(0,2,3)$

Gray Codes

- Single bit change from one code value to the next
- Standard gray code – formed by reflection

Logic Gates



- **Fan-in**: The number of inputs of a gate.
- **Complete set of logic**: Any set of gates sufficient for building any boolean function.
  e.g. {AND, OR, NOT}
  e.g. {NAND} *(self-sufficient / universal gate)*
  e.g. {NOR} *(self-sufficient / universal gate)*
- **SOP expression** – implement using 2-level AND-OR circuit or 2-level NAND circuit
- **POS expression** – implement using 2-level OR-AND circuit or 2-level NOR circuit
- **Programmable Logic Array (PLA)**: 2-level AND-OR array that can be "burned" to connect

Karnaugh Maps

- **Implicant**: Product term with all '1' or 'X', but with at least one '1'
- **Prime implicant**: Implicant which is not a subset of any other implicant
- **Essential prime implicant**: Prime implicant with at least one '1' that is not in any other prime implicant
- **Simplified SOP expression** – group '1's on K-map
- **Simplified POS expression** – find SOP expression using '0's on K-map, then negate resulting expression